Parallel Internally Contracted Multireference Configuration Interaction

ABIGAIL J. DOBBYN,[1] PETER J. KNOWLES,[2] ROBERT J. HARRISON[3]

[1]*Department for Computation and Information, Daresbury Laboratory, Daresbury, Warrington, Cheshire WA4 4AD, UK*
[2]*School of Chemistry, University of Birmingham, Edgbaston, Birmingham, UK*
[3]*Pacific Northwest National Laboratory, Richland, Washington*

Received 5 January 1998; accepted 26 February 1998

ABSTRACT: A parallel implementation of the internally contracted (IC) multireference configuration (MRCI) module of the MOLPRO quantum chemistry program is described. The global array (GA) toolkit has been used in order to map an existing disk-paging small-memory algorithm onto a massively parallel supercomputer, where disk storage is replaced by the combined memory of all processors. This model has enabled a rather complicated code to be ported to the parallel environment without the need for the wholesale redesign of algorithms and data structures. Examples show that the parallel ICMRCI program can deliver results in a fraction of the time needed for equivalent uncontracted MRCI computations. Further examples demonstrate that ICMRCI computations with up to 10^7 variational parameters, and equivalent to uncontracted MRCI with 10^9 configurations, are feasible. The largest calculation demonstrates a parallel efficiency of about 80% on 128 nodes of a Cray T3E-300.
 J Comput Chem 19: 1215–1228, 1998

Keywords: configuration interaction; parallel computation; quantum chemistry; electronic structure; electron correlation

Presented in part at the HPCI Conference, University of Manchester Institute of Science and Technology, January 1998

Correspondence to: P. J. Knowles

Contract/grant sponsor: EPSRC; contract/grant number: GR/K4656

Contract/grant sponsor: EEC; contract/grant number: FMRX-CT96-088

Contract/grant sponsor: U.S. Department of Energy with Battelle Memorial Institute; contract/grant number: DE-AC06-76RLO

 CCC 0192-8651 / 98 / 111215-14

Introduction

The theoretical study of chemical reactions can provide a great deal of useful information; for example, reaction rates, product state distributions, and insights into reaction mechanisms. Such research is relevant to all areas of chemistry, being of fundamental importance in the modeling of processes occurring in combustion and in the atmosphere. As the first step in such studies, it is often necessary to obtain highly accurate global potential-energy surfaces (PESs), or at least definitive information about the height of reaction barriers and reaction energies.

For dynamical calculations to be worthwhile, the information about the internuclear interaction energies must be of chemical accuracy. This means that a large proportion of the static, as well as the dynamic, electron correlation energy must be included. In the case of PESs, this must be done with uniform accuracy. Therefore, it is necessary to use multireference configuration interaction (MRCI) methods, which, in contrast to other methods that utilize only a single reference configuration, are capable of describing excited states, near-degeneracy effects, and multiple open-shell systems, as well as the two-electron cusp. These methods are inevitably very costly in all aspects of computing resources. As well as involving large numbers of arithmetic operations, very large intermediate data structures must be manipulated, demanding large central memory and peripheral storage, with fast movement of data between them. A further consideration is that PESs must be global, so that they cover not only the interaction region, but extend well into both the entrance and exit channels. To achieve this, very many points have to be calculated, of the order of 1000 for a triatomic system.

The increasing availability and performance of parallel computers[1,2] offers the possibility of at least partially satisfying the very heavy resource demands of MRCI computations. The use of parallel computers in quantum chemistry is not new,[3–5] although the majority of work done in the past has concentrated on self-consistent field (SCF) programs, with more accurate methods being addressed more recently [3–8]. There have been several previous presentations of parallel MRCI programs [4–7], the most recent of which is that forming a part of the COLUMBUS program package.[6,7] The parallelization is there carried out across pairs of segments of the residual and coefficient vectors, which are dynamically assigned to the processors. This implementation has shown excellent scaling, and is also fully portable to many different machines.

Unfortunately, all of these parallel MRCI implementations suffer from the use of an underlying ansatz, which is much more computationally expensive than it needs to be. Following the suggestion of Meyer,[9] the internally contracted MRCI (ICMRCI) approach[10–14] has emerged as the method of choice for MRCI computations, particularly where the number of reference configurations is large. Instead of using the standard uncontracted basis of all configuration state functions (CSFs) which are related to any reference configuration by at most two excitations, the ICMRCI basis consists of orbital excitations applied to the complete multideterminant-reference wave function. As a result, the size of the basis does not depend on the number of reference configurations, and is usually several orders of magnitude smaller than the corresponding set of uncontracted CSFs. The errors introduced by this contraction procedure have been shown to be very small, and the only drawbacks in adopting the approach derive from the complex nature of the basis functions generated by the contraction procedure. Individual matrix elements are more expensive to calculate than in the uncontracted case (although there are far fewer of them), and the Hamiltonian matrix formulas (and thus the software used to compute them) are more complicated. However, we note that the dominant computational work in uncontracted MRCI, namely the interactions between doubly external CSFs, for which an efficient parallel implementation has been reported,[7] almost totally disappears in ICMRCI. In this study, we report a parallel implementation of ICMRCI, together with comparison, where possible, with standard uncontracted MRCI.

It is reasonable to suggest for the construction of PESs, which require calculations at very many different geometries, that the best strategy would be a data farming one, whereby different geometries are calculated on each processor. We have pursued this possibility specifically for the IBM SP2 parallel computer, with individual points on the surfaces being calculated on separate processors of the computer. The SP2 is superbly equipped for these calculations: the single processor performance is excellent, and there are generally large scratch disks attached to each processor, which makes the large amount of I/O involved not only

possible but also fast. This, however, is not a sensible approach for the other types of parallel computers, such as the Cray T3D/T3E, which historically have not only had relatively poor I/O rates, but also had fairly limited amounts of disk space. Additionally, it should be possible to perform much larger calculations if all of the memory resources of the entire parallel machine could be devoted to a single computation; the task-farming approach is inherently wasteful in this respect, and takes no advantage of the sophisticated interprocessor data transfer facilities available on the latest parallel supercomputers. For these reasons it is necessary to develop a distributed data, or truly parallel, version of the MRCI code.

Our aim is therefore to develop a scalable and portable parallel MRCI program. In balance with these aims, particularly the first, is a pragmatic approach, which attempts to minimize the changes between the sequential and parallel codes. This should make the development quicker and easier, as well as reducing the chance of introducing errors into the code. Furthermore, it is important to realize that the MOLPRO program system[15] and the MRCI program within it are continually evolving, so that it is essential that the parallel version of the code be easily maintained. Given the wide variety of different parallel computers that are routinely used at present it is highly desirable that the parallel code should be portable to a large number of different platforms (as the sequential version of MOLPRO already is). This is ensured by the use of the Global Array subroutine libraries,[16] together with TCGMSG,[17] which are portable software toolkits for the development of parallel programs.

Contracted Multireference Configuration Interaction

To carry out contracted multireference CI computations within our program suite, MOLPRO,[15] four distinct tasks can be identified:

1. Calculation of one- and two-electron Hamiltonian integrals, and their sorting into canonical order for use by the other program stages.
2. Self-consistent field calculation (SCF) to obtain molecular orbitals for use as starting guesses in the next program stage.
3. Multiconfiguration self-consistent field calculation (MCSCF), usually within the complete active space (CASSCF) scheme,[18] which is carried out to optimize the molecular orbitals for that reference space.[19,20]
4. Iterative direct diagonalization of the MRCI Hamiltonian matrix.

The evaluation and sorting of the two-electron integrals program stage has been parallelized, but this will not be described further here. Stages 2 and 3, the SCF and MCSCF, have not been parallelized. For the calculation of PESs, parallelization of stage 2 is not necessary as orbitals from an adjacent geometry can be used as starting guesses in the MCSCF program, which itself will converge rapidly because the neighboring-geometry orbitals will be close to the solution. In the case in which only a few important geometries are being investigated, for example, at transition states or asymptotes, MCSCF orbitals can be obtained conveniently on a scalar computer, and so again parallelization of the MCSCF program is not an issue.

Before giving any description of the parallelization of the code, we review some aspects of ICMRCI theory and its implementation in MOLPRO. In the following, the indices $i, j, k, \ldots$ are used to denote ''internal'' or ''active'' orbitals, which are occupied in at least one of the reference configurations, and out of which excitations are made in the CI wave function. The letters $a, b, c, \ldots$ will denote ''virtual'' or ''external'' orbitals, which are unoccupied in all reference configurations, but of which up to two orbitals can be filled in configurations making up the CI wave function.

The wave function is represented in a basis of configurations: the internal configurations, Ψ_I, which are CSFs connected to at least one reference configuration by at most a double excitation involving valence orbitals alone; ''singly external'' CSFs, Ψ_S^a, containing exactly one singly occupied external orbital; and contracted pair-external configurations, Ψ_P^{ab}, formed by a double excitation from two active orbitals into two virtual orbitals, acting on the reference wave function as a single entity:

$$\Psi = \sum_I c^I \Psi_I + \sum_S \sum_a c_a^S \Psi_S^a + \sum_P \sum_{ab} C_{ab}^P \Psi_P^{ab} \quad (1)$$

S and P represent internal $N-1$ and $N-2$ electron hole states (where there are N electrons), and will be referred to as singles and pair functions; there are N_S and N_P of these, respectively. There is a very important distinction between the singly and doubly external configurations in our imple-

mentation; the singly external configurations are simple configuration state functions, as used in other MRCI approaches, but the double externals, of which the uncontracted counterpart form the most numerous set, are contracted, and therefore comprise a much smaller set, with each contracted configuration formally a linear combination of many CSFs. Because the number of contracted doubly external configurations does not increase with the size of the reference space, calculations with very large reference spaces are possible. Furthermore, this makes the internally contracted ansatz much more computationally efficient than MRCI programs, which do not use contracted pairs. It has been shown that this approximation has very little effect on the energy, and the one-electron properties of the wave function.[11]

The doubly external configurations are defined precisely as:

$$\Psi_P^{ab} = \frac{1}{2}\left(\hat{E}_{ai,bj} + p\hat{E}_{bi,aj}\right)\Psi_0 \qquad (2)$$

where Ψ_0 is the set of reference configurations, that is, the total reference wave function, and $p = \pm 1$ represents the spin coupling (singlet or triplet) of the pair of external orbitals. Thus, the $N-2$ electron hole states, the pair functions P, are uniquely identified with the label ijp, and these, together with the configurations Ψ_P^{ab}, are ''internally contracted.'' The contracted doubly external configurations are not orthogonal, and the coupling coefficients that must be evaluated between the configurations are rather complicated, thus making the implementation of the internally contracted MRCI somewhat more involved than the uncontracted MRCI.

It is possible to use a set of similarly contracted singly external configurations; however, the increase in computational effort associated with their orthogonalization, and increased complexity of the coupling coefficients, probably outweighs any advantage in doing so; the number of uncontracted singly-external configurations will always be much smaller than the number of uncontracted doubly external configurations. In a typical large calculation there will be $O(10^3)$ reference configurations, $O(10^3$–$10^4)$ internal configurations (N_I), $O(10^3$–$10^4)$ singles functions (N_S), $O(10^2)$ pair functions (N_P), $O(10^2)$ external molecular orbitals (N_{ext}), and $O(10)$ internal molecular orbitals. The size of the total contracted configuration basis will be of the order of 10^6, whereas the corresponding uncontracted configuration basis will be of the order of 10^7–10^8.

Within this basis the Schrödinger equation is solved using an iterative method; a modified Davidson procedure is used here.[11,21] An iterative method is used because it is not possible to hold the whole of the Hamiltonian matrix in memory, as well as the fact that usually only the lowest one or two eigenvalues and eigenvectors are required. At each iteration the wave function is represented by a relatively small set of trial vectors, $\mathbf{c}$, which are generated by the operation of the Hamiltonian on the wave function, $\mathbf{g} = \mathbf{Hc}$. Each iteration, therefore, consists of the calculation of the residual vector, $\mathbf{g}$, followed by the update of $\mathbf{c}$, in preparation for the next iteration. This procedure is repeated until the wave function has converged and the trial vectors span the subspace in which the eigenvector lies.

The evaluation of $\mathbf{g}$ is the most computationally intensive part of the MRCI program, and we will therefore give an outline of what this involves, as it will be this part that will require the most attention on parallelization of the code. The equations will not be presented in detail and, for further information, the reader is referred to refs. 11–13. The trial vector is represented in the basis of configurations, $|\Psi_Z\rangle$, and in the residual vector $\mathbf{g}$ we can consider separately those elements of the internal, singly external and doubly external configurations:

$$\sum_Z g_Z|\Psi_Z\rangle = \sum_I g_I|\Psi_I\rangle + \sum_S \sum_a [\mathbf{g}_S]_a|\Psi_S^a\rangle + \sum_P \sum_{ab} [\mathbf{g}_P]_{ab}|\Psi_P^{ab}\rangle \qquad (3)$$

where $\mathbf{g}_S$ is a vector with elements a, and $\mathbf{g}_P$ is a matrix with elements ab. We can write down expressions for these constituents of $\mathbf{g}$:

$$g_I = \left\langle \Psi_I \middle| \hat{H} \middle| \sum_{I'} c^{I'}\Psi_{I'} + \sum_S \sum_a c_a^S \Psi_S^a + \sum_P \sum_{ab} C_{ab}^P \Psi_P^{ab} \right\rangle = w_{I(I')} + w_{I(S)} + w_{I(P)} \qquad (4)$$

$$[\mathbf{g}_S]_a = \left\langle \Psi_S^a \middle| \hat{H} \middle| \sum_I c^I \Psi_I + \sum_{S'} \sum_a c_a^{S'} \Psi_{S'}^a + \sum_P \sum_{ab} C_{ab}^P \Psi_P^{ab} \right\rangle = [\mathbf{w}_{S(I)}]_a + [\mathbf{w}_{S(S')}]_a + [\mathbf{w}_{S(P)}]_a \qquad (5)$$

$$
\begin{aligned}
[\mathbf{g}_P]_{ab} &= \left\langle \Psi_P^{ab} \middle| \hat{H} \middle| \sum_I c^I \Psi_I + \sum_S \sum_a c_a^S \Psi_S^a \right. \\
&\quad \left. + \sum_{P'} \sum_{ab} C_{ab}^{P'} \Psi_{P'}^{ab} \right\rangle \\
&= [\mathbf{w}_{P(I)}]_{ab} + [\mathbf{w}_{P(S)}]_{ab} + [\mathbf{w}_{P(P')}]_{ab} \quad (6)
\end{aligned}
$$

We thus recognize the various ''interactions'' between different types of configurations, which make contributions to $\mathbf{g}$. In general, $w_{X(Y)}$, where X and Y can be anyone of I, S, or P, can be written as:

$$
w_{X(Y)} = \sum_Y \sum_\lambda f_\lambda \alpha_\lambda(X, Y) c^Y + \text{other terms} \quad (7)
$$

f_λ is a set of two-electron integrals $(rs|tu)$, with the index λ running over zero, one, two, three, or four internal molecular orbitals (MOs). Each set of two-electron integrals f_λ will consist of all those two-electron integrals whose full label $rstu$ contains the partial label λ. For example, when the index λ runs over one internal MO, the set f_λ will consist of all two-electron integrals containing the specified internal MO and any three external MOs; when the index λ runs over three internal MOs, the set f_λ will consist of all two-electron integrals containing the three specified internal MOs and any one external MO. The ''other terms'' will be made up of one-electron integrals and the relevant coefficients and, in some cases, coupling coefficients; these terms are not computationally expensive and will not be discussed further. Of course, $w_{X(Y)}$ and c^Y will be scalars, vectors, or matrices with dimension equal to the number of external orbitals, depending on whether X and Y are I, S, or P. One could also think of f_λ as being a tensor of rank 0, 1, 2, 3, or 4, depending on whether the index λ runs over 4, 3, 2, 1, or 0 internal MOs; the rank of f_λ will match the rank of c^Y to produce $w_{X(Y)}$ with the correct rank. Depending on the particular identity of X and Y, the matrices f_λ and c^Y will then be combined in inner or outer tensor products to form the residual w. The coupling coefficients, $\alpha_\lambda(X, Y)$, are not held in a ''formula file,'' but are evaluated as needed from a minimal set of symmetric group representation matrices, which are set up at the start of the calculation[12] and held in local memory on each processor; the amount of memory required depends strongly on the case, but is typically between 10^5 and 10^6 words.

From the just discussed schematic overview of the equations involved in the MRCI program we can see that it is necessary to store the following quantities: one- and two-electron integrals $(N_{\text{basis}}^4/8)$, where N_{basis} is the number of atomic/molecular orbital (AO/MO) basis functions; c^I and g_I (N_I); $\mathbf{g}_S$ and $\mathbf{c}^S$ $(N_S \times N_{\text{ext}})$; and $\mathbf{g}_P$ and $\mathbf{c}^P$ $(N_P \times N_{\text{ext}}^2)$. These quantities are too large for all of them to be held simultaneously in the core memory. They are therefore stored as a set of scratch files. Each file is direct access and word addressable but is organized in records of any length. All input and output (I/O) to these scratch files is through a set of FORTRAN subroutines: *reserv*, which reserves space for a record of a specified length on the specified file; *lesw*, which reads a one-dimensional array of double precision data of particular length from an offset from the start of the record on a specified file; *sreibw*, which writes in an analogous fashion. These FORTRAN subroutines implement buffering, and perform raw read and write operations through a set of lower level C subroutines. The files are generally scratch files; that is, they are deleted at the end of a calculation, but can individually be made permanent or ''named,'' so that quantities such as the molecular orbitals can be saved from one calculation to the next.

The normal algorithms for calculating each $w_{X(Y)}$ recognize, first, that the coupling coefficients, $\alpha_\lambda(X, Y)$, are extremely numerous, and in some cases very sparse, and so they cannot conveniently be stored in full in memory; thus, for a given interaction, the calculation is driven by the sequential computation of $\alpha_\lambda(X, Y)$, combining with all other quantities that are held in full in memory. In some cases, however, either when a very large calculation is being attempted, or if a more moderate calculation is carried out on a computer with only a relatively small amount of core memory, it is not possible to hold the whole of any one of the coefficient segments, c^Y or g^X, in core memory. Therefore, the calculation of $w_{X(Y)}$ has to be divided up in some way, so that it is essential to use algorithms such that all of $w_{X(Y)}$, c^Y, and f_λ are read in batches if necessary. The code will then have the general structure shown in Table I for the evaluation of $w_{X(Y)}$ (for λ going over just one number of internal MOs).

Notice that the inner part of the algorithm can generally be implemented very efficiently as a sequence of two vector–vector, matrix–vector, or matrix–matrix operations. The order of the multi-

TABLE I.
Generic Batching Algorithm for MRCI.

check on amount of memory available
consideration of memory required to hold $w_{X(Y)}$, c^Y and f_λ as well as work space
decision on batching scheme
do $ibatch_x = 1, nbatch_X$
  zero (if first contribution) or read in g_X for X in batch $X_1{:}X_2$
    do $ibatch_Y = 1, nbatch_Y$
      read in c^Y for Y in batch $Y_1{:}Y_2$
      do $ibatch_\lambda = 1, nbatch_\lambda$
        read in f_λ for λ in batch $\lambda_1{:}\lambda_2$
        calculate $\alpha_\lambda(X,Y)$ for X_1 to X_2 and Y_1 and Y_2
        for X in batch $X_1{:}X_2$
          $g_X = g_X + \sum_{Y=Y_1}^{Y_2} \sum_{\lambda=\lambda_1}^{\lambda_2} f_\lambda \alpha_\lambda(X,Y) c^Y$
        end do
      end do
  write g_X for X in batch $X_1{:}X_2$ to file
  end do
end calculation of $w_{X(Y)}$

plications is not necessarily as shown, but will be chosen to minimize the operation count.

The order in which the loops in Table I are implemented will vary, as will the relative size of the batches, $nbatch_X$, $nbatch_Y$, and $nbatch_\lambda$, depending partly on the relative sizes of $w_{X(Y)}$, c^Y, and f_λ. In general, an attempt is made to hold the smallest quantities completely in memory and to batch over the larger quantities, preferably just once, so that each is read just once through. There are cases where there are quite different algorithms depending on the amount of memory available, and therefore the amount of batching required.

We have given above a fairly detailed overview of the memory management aspects of the MRCI program in its scalar version as it existed before this research began. The purpose of doing this rests on the recognition that the general model of paging to and from disk, adopted for the small-memory workstations of the last decade, maps quite closely to the situation prevailing with many modern parallel computers, where for each processor there is a relatively small amount of fast local memory, with slower access to storage on other processors. This nonuniform memory-access (NUMA) programming model unites sequential and parallel programming, and simplifies the design of scalable parallel algorithms. In the following section, we describe how these original paging algorithms can therefore be used in the parallel context for the effective distribution of data without wholesale redesign of a complicated code.

Parallel Implementation

The sequential code just described is seen to rely heavily on the use of scratch files, and therefore on the availability of fast I/O and relatively large amounts of disk space. It is clear that such reliance on I/O will be impossible when using massively parallel processing (MPP) machines, which to date have typically shown extremely poor I/O capabilities (of which the performance seldom scales with the number of processors), and often have relatively little hard disk available.

Therefore, the first step necessary for the development of a scalable program to run on a MPP computer is the elimination of the I/O and the removal of the scratch files. This we do by replacing the reading and writing to scratch files by reading and writing to global memory; that is, the sum of the memory of all the participating processors. So, for example, all the two-electron integrals will be held in memory, with a roughly equal number of them stored on each processor. This is straightforwardly implemented using the Global Array subroutine library.[16]

GLOBAL ARRAYS

The global array (GA) subroutine library implement a portable ''shared-memory'' programming model for distributed-memory computers, developed specifically for use in parallel quantum-

chemistry codes. In evaluating the advantages of using such a library and its underlying programming model, it is worth considering the two main alternatives: a shared-memory model, and a message-passing model. The main advantage of using a shared memory model is the ease of programming, as well as the fact that the one-sided communication does not require synchronization between processors, which facilitates load-balancing strategies. The disadvantage of such a model is, however, that it is not always obvious to the programmer whether or not they are using nonlocal data; that is, data not held on that processor, so that the extra cost of obtaining this data is not fully accounted for. In contrast, the main advantage of a message-passing paradigm is that it is very clear when nonlocal memory is being accessed, which in fact leads to the main disadvantage, which is the difficulty of designing such programs, as well as the requirement for explicit cooperation between processors, and the synchronization which must necessarily occur (though the impact of this can be reduced to some extent by the use of asynchronous communication and buffering of data).

The global array subroutine libraries take the advantages of both these models: they allow for ease of programming, and lack of synchronization between processors, while acknowledging that the nonlocal data take more time to access. Global arrays can be created and destroyed using the GA functions *ga_create* and *ga_destroy*, respectively, whereas data can be read from, written, and accumulated to a global array with the GA subroutines *ga_get*, *ga_put* and *ga_acc*.

The GAs are implemented using the fastest native communication, and interoperate with MPI[22] and TCGMSG.[17] MPI and TCGMSG provide basic message-passing subroutines, such as send and receive, as well as global operations, and TCGMSG includes a shared counter (*nxtval*), which is a useful tool for dynamic load balancing. The *nxtval* counter has been used extensively in the parallel implementation and, in a few cases, the message passing and global operations have also been found useful.

GA WITHIN MOLPRO

To develop a completely replicated version of the code (i.e., where all processors do exactly the same work) the records on each of the files are replaced by global arrays. The substitution of the real I/O for reading and writing to the global arrays is simple to carry out, and can be done within the existing FORTRAN subroutines that manage the I/O, without any changes whatsoever to the rest of the code. Therefore, within the *reserv* subroutine there is now a call to the GA function *ga_create*; within *lesw* there is a call to *ga_get*, and within *sreibw* there is a call to *ga_put*. In this initial replicated data implementation all processors read the data they require, whereas the processors only write the data that they ''own'' (which can be determined using the GA subroutine *ga_locate_region*), after which they synchronize. When we wish to save the scratch files, it is possible to ''name'' them, in which case they are then treated as a real file and written to disk. In this case, only the zeroth processor reads and writes to the file, and broadcasts the data to the other processors. This facility is particularly useful for restarting the program from previously calculated molecular orbitals. This replicated data mode of the code is denoted by the variable *mpp_state* having the value one.

Obviously, in order to develop a parallel version of the code, it is necessary to go beyond this replicated data approach to a distributed data approach, where processors work on complementary data, so that each processor needs to read and write distinct pieces of data. In this case, each processor simply reads and writes the data on which it is working, without reference to any other processor. At present, this type of I/O does not work with named (i.e., real) files. This distributed data mode of the code is denoted by the variable *mpp_state* having the value of 2. At the start and end of each stage of the program, where there is a call to *sreibw* with *mpp_state = 2*, the processors synchronize.

The parts of the code that have not been parallelized at all (e.g. the SCF) will run exclusively within the *mpp_state = 1* mode. The MRCI part of the code will have within it both the *mpp_state = 1* and the *mpp_state = 2* modes. When it is reading and writing to the global arrays in the *mpp_state = 1* mode, this does not necessarily mean that there has been no work done in parallel, but may mean that the work has been divided up across the tasks which are required to evaluate some length of data, rather than across the length of data itself, which will be the case in the distributed data mode. To facilitate such a type of parallelism, a new FORTRAN I/O subroutine is introduced, *accw*, which will in these cases replace a call to *sreibw*. In this subroutine, the data passed to it is first summed globally and then the completed data is written to the appropriate global array. In

cases when it is not the first contribution to **g** that is being formed, there will more than just a change from *sreibw* to *accw* required. In order that the original value of **g** is not also summed, it is not read in at the start of the subroutine, but always zeroed, and the original value of **g** is read in and added to the new contribution after the latter has been summed. In fact, each processor does not need access to the whole of the completed data, but only the part which it ''owns.'' Therefore, it is found to be more efficient here to use a message-passing routine such that each processor receives that data that it requires from all the other processors, and performs the necessary summation, before accumulating the data to the global array.

At the start of the calculation it is necessary to divide the total amount of allocated memory between that used for the global arrays (i.e., global memory) and that used as local memory. This division can be altered between program steps, which is particularly useful after the sort of the two-electron integrals, which requires quite large amounts of global memory. In practice, the amount of global memory allocated during the MRCI program stage is typically double that necessary to hold the two-electron integrals. The GA subroutine libraries include a local memory management library called MA, which is used by the MPP version of MOLPRO.

The above constitutes a complete framework from which we can proceed to develop a truly parallel version of the code, in which the work, and where possible the memory requirements, of the program are divided up over the processors. In particular, a mechanism for the progressive introduction of parallel code has been set up. An initial replicated version of any code, using $mpp\_state = 1$, can be gradually transformed into a parallel code through successive replacement of code segments, using $mpp\_state = 2$ where necessary for supporting distributed data structures.

ANALYSIS OF COMPUTATION AND ASSIGNMENT OF TASKS

As has been stated previously, the calculation of the residual vector **g** is the most computationally intensive part of the code. Typically the contribution to the pair–single [$\mathbf{w}_{P(S)}$] and single–pair interactions [$\mathbf{w}_{S(P)}$] from λ running over three internal MOs takes the most time to evaluate of all the constituent parts of **g**, although the pair–pair [$\mathbf{w}_{P(P')}$] interaction, and the single–single [$\mathbf{w}_{S(S')}$] interaction, can also take significant amounts of time. The exact breakdown of the total time to evaluate **g** is case dependent.[11] The other interactions, such as the internal–pair, pair–internal, internal–single, single–internal, and internal–internal, as well as the work involved in the initialization and the calculation of the density matrix account for around 5% of the total CPU time. Although this may not seem significant, a consideration of Amdahl's law makes it quite clear that this work must also be parallelized if reasonable speedups are to be obtained using more than a small number of nodes. For example, if a program running on 64 processors has 95% of its work parallelized, the best speedup it can hope to obtain is 15.4 (i.e., only 24% efficiency). It is the fact that the total time for the MRCI program step is not overwhelmingly dominated by any one interaction, or distinct set of localized tasks, which must limit our expectations of obtaining excellent speedups.

At this stage it is now necessary to consider how the work involved in these calculations can be divided up over the processors (i.e., the division of the work into separate tasks and the assignment of these tasks to individual processors).

As mentioned earlier in this work, we hope normally to take advantage of the considerable work that has already gone into the paging strategies used throughout the program, so that, for example, one might, in the scheme of Table I, envisage each processor carrying out the work of a single batch (i.e., each task will be associated with a batch). Generally speaking, the work should be parallelized across just one set of batches (i.e., $nbatch_X$, $nbatch_Y$, or $nbatch_\lambda$), thus leading to the complete distribution of one of these quantities. It could be argued that the parallelization is best carried out over $nbatch_X$, as this results in the distribution of the output, and thus avoiding the need for a summation of the result in *accw* over the processors.

There are several advantages in using the batching schemes of the sequential program. First, the work of the development of the parallel code is greatly reduced as the mechanism to divide up the work, as well as the program infrastructure to deal with these batches, already exists in the code. Second, this approach simplifies the decision on how the work should be divided up, as a careful analysis has already been done in the sequential code as to the most efficient way to batch up the work. Third, the changes between the sequential and parallel codes should, by this method, be kept to a minimum. It should be noted, however, that

in certain cases there may be other optimal ways to divide up the work in the parallel version of the program, which differ from the batching existing in the original sequential code.

In the design of any parallel algorithm, and in the setting up and allocation of tasks here, there are two main important factors that must be taken into account.

Factor 1. The minimization of overheads (latency and bandwidth) associated with communication between processors, which corresponds here to the mindful use of accesses to the global arrays and, where possible, the maximization of use of local data. Ideally, all data structures would be replicated on all processors, but, particularly for large MRCI calculations, the size of many of these data objects exceeds local memory, and they must be distributed. This consideration will be particularly critical on machines where the bandwidth is low. On some machines, where the latency for communication is high, it is necessary to ensure that the number of GA calls is minimized, so that there are a few accesses to global arrays with large chunks of data, rather than many accesses with small amounts of data.

Factor 2. The load balancing has to be optimized, so that each processor does an equivalent amount of work. In some situations where the tasks are known to take similar lengths of time, this can be done statically, using expressions such as:

$$if(\,ga\_nodeid().eq.mod(icount, ga\_nnodes()))\ then$$

where *ga_nodeid*() is the label of that particular processor, *ga_nnodes*() is the number of processors, and *icount* is a count on the number of tasks. However, in other situations there might be no straightforward model available to predict the work involved in each task, or it might be known that the tasks vary a great deal in the work associated with them. Here it will be necessary to use a dynamic load balancing strategy, with the help of a shared counter. In this case, attention must be given to the size of the individual tasks, and/or the number of tasks carried out for every call to the counter *nxtval*. The latter point is particularly important on certain machines where calls to *nxtval* can be quite costly, especially when one considers that all calls to *nxtval* are dealt with by one processor. To ensure that the use of the shared counter does not become a bottleneck in the code, $ga\_nnodes() \times T_{nxtval}$ must be much smaller than the time to execute the tasks completed for each call to *nxtval*, where T_{nxtval} is the cost of each call. Another factor which is particularly important for the dynamic load balancing, but is also relevant to the static load balancing, is the total number of tasks available. For static load balancing this should either be very many times larger than the number of processors, or if there are not so many tasks, it should be a simple multiple of the number of processors. For dynamic load balancing to function properly there must be several times more tasks than processors, the actual ratio depending on how disparate the task times are. Considering the particular case of the MRCI program in MOLPRO, and taking into account the relative number of pair and single functions, and the number and size of f_λ, it is apparent that the best load balancing will be achieved if work is distributed across the single functions rather than across the pair functions, but in the case that the index λ is running over 1 or 0 internal MOs, it may well be best to parallelize over the elements of f_λ.

EXPLICIT EXAMPLES OF PARALLEL IMPLEMENTATION

We now illustrate our parallel implementation by considering some examples of the most important interactions in turn, and discussing the methods used to divide up the work and, where necessary, the data, over the processors.

We start with the contribution to the pair−single and single−pair interaction made by the index λ running over three internal MOs. These two interactions are treated together, because the coupling coefficients, $\alpha_\lambda(P, S)$ and $\alpha_\lambda(S, P)$, are equivalent. The number of singly external integrals f_λ is small, so these are held in full in memory. The code contains a pair of loops, the outer being over batches of pair functions, and the inner over batches of single functions. In the sequential code, the number of batches over the pairs is minimized, so that most of the batching is over the singles. This then fits well with the parallel version where the work is divided over batches of singles functions. Any attempt at static load balancing gives very poor performance; the cost of computing the coupling coefficients, $\alpha_\lambda(S, P)$, is very far from uniform, depending strongly on factors such as the number of spin couplings associated with a given orbital occupation in S. Accordingly, we divide the work into a number of tasks each of which is expected to take roughly the same time according to an estimated operation count, but then apply dynamic load balancing with a shared counter, as

described earlier. The total number of tasks is $num\_tasks = ga\_nnodes() \times nspread\_sin$, where $nspread\_sin$ is a parameter, which can be tuned to improve performance; a value of 3 is used, so there are, on average, three tasks for each processor. Remaining imbalance between tasks is ameliorated by ordering the tasks in decreasing order of expected cost. From this description of the parallel work, it can be seen the $\mathbf{g}_S$ can be written in a distributed fashion to the appropriate global array with $mpp\_tstate = 2$, whereas a call to *accw* will be required for $\mathbf{g}_P$, so that the contributions calculated on each processor to $\mathbf{g}_P$ can be summed.

We next consider the contribution to $\mathbf{w}_{P(P')}$ made by the index λ running over 0-internal MOs; i.e., that is, integrals of the type $(ab|cd)$. In this case, the coupling coefficients are simply $\delta_{PP'}$, so that the interaction is diagonal in the internal pair functions. In our code, this interaction is computed in the atomic-orbital basis.[10,11] The work is carried out in three distinct stages: transformation of $\mathbf{c}^P$ from the molecular- to the atomic-orbital basis; multiplication of $\mathbf{c}^P$ by f_λ (in atomic-orbital basis) to form $\mathbf{g}_P$; backtransformation of $\mathbf{g}_P$ from the atomic- to the molecular-orbital basis. In the sequential code, as many pair functions as possible are held in memory (i.e., the number of batches of pairs is minimized), while the integrals are read in many relatively small batches. One approach to parallelization of this subroutine is to distribute both $\mathbf{c}^P$ and $\mathbf{g}_P$ over the processors, which is possible here as the interaction is diagonal. There are two main disadvantages to such an approach. The first is that the number of pair functions is generally not large, so that load balancing may be a problem. The second is that the number of two-electron integrals (f_λ) is very large [$O(N_{\text{ext}}^4)$] and these will have to be read by *every* processor. We have instead followed an approach in which the integrals and operations on them are distributed. All c^P and g^P are stored on each processor, and the transformations between MO and AO bases are divided statically across the pairs, with appropriate merging at the end of each stage. The dominant CPU work, involving multiplication with integrals, is then divided dynamically, with each processor obtaining just the integrals it needs from the global arrays. In principle, all communication could be avoided by allowing each processor to work on exactly the set of integrals which are locally resident as part of the global array; we have not explored this possibility, because, if used naively, it eliminates the ability to dynamically load balance. The disadvantage of this scheme is that $\mathbf{c}^P$ and $\mathbf{g}_P$ cannot be distributed over the processors, so that if there is limited memory available, or there are many pair functions, $nbatch_P$ may be greater than one. The consequence of this is, first, that the two-electron integrals will have to be read several times and, second, that there may be only a few pair functions available to divide up over the processors, adversely affecting the load balancing in the first and last stages.

The remaining contribution to $\mathbf{w}_{P(P')}$, made by the index λ running over 0 and 2 internal MOs, is discussed next. In this subroutine, the batches over $\mathbf{g}_P$ form the outermost loop, followed by: the evaluation of the coupling coefficients $\alpha_\lambda(P, P')$ for *all* values of P'; the batches over the index λ, (i.e., the doubly external two-electron integrals f_λ); and, finally, the batches over $\mathbf{c}^{P'}$. There are then two inner loops over the $\mathbf{c}^{P'}$ in the batch and then $\mathbf{g}_P$ in the batch. In the sequential version the number of batches over $\mathbf{g}_P$ and λ are minimized, so that there will be the most batching over $\mathbf{c}^{P'}$. The parallelized version follows this scheme, by dividing the work across $\mathbf{c}^{P'}$, except in the case where there are not enough tasks, and in this case the work is divided across $\mathbf{c}^{P'}$ and $\mathbf{g}_P$; that is, both of the innermost loops, so that $\mathbf{c}^{P'}$ is only partially distributed. The number of loop elements in each task is, in general, given by:

$$N_{\text{ele}} = \max((N_{P'} \times num\_batch_P)/(ga\_nnodes() \times nbalan_P), num\_batch\_min_P)$$

where $nbalan_P$ and $num\_batch\_min_P$ are parameters tuned to improve the load balance, and num_batch_P is the number of pair functions in the batch of $\mathbf{g}_P$. However, when N_{ele} is greater than num_batch_P, it is set to $N_{\text{ele}} = \text{int}(N_{\text{ele}}/num\_batch_P) \times num\_batch_P$. Values of 4 and 2 are used for $nbalan_P$ and $num\_batch\_min_P$ respectively. Once every element of these two loops has been computed, a call to *accw* is necessary for $\mathbf{g}_P$, so that the contributions calculated on each processor to $\mathbf{g}_P$ can be summed. The doubly external two-electron integrals will be read in $nbatch_P$ times (number of $\mathbf{g}_P$ batches), the number of which is kept to a minimum.

It is also necessary in this subroutine to divide up the evaluation of the coupling coefficients, $\alpha_\lambda(P, P')$, and this is done dynamically over $P(\mathbf{g}_P)$, and must be followed by a global sum so that all processors have a copy of these. This is an example of parallelization over an intermediate, which needs to be summed before being able to proceed with further computation, and of course implies a

synchronization of the processors. A frequent situation in code is that there is often fairly appreciable amounts of work required to calculate intermediates, which are then used for all values of g_X and/or c^Y, rather than all of the parallel work being over the evaluation of g_X directly.

The above examples show how effective parallelization of the ICMRCI code has been achieved; there are other parts of the code that have not been described, but which have been parallelized using similar ideas.

Example Applications and Demonstration of Scaling

We have carried out several different benchmark calculations, the details of which are presented in Tables III–VI, on both SP2 and Cray T3D/T3E MPP machines. Table II gives an approximate guide to the capabilities of the machines used. Of course, the processor speeds and bandwidths, etc., relate to the theoretical peak perfor-

TABLE II.
Characterization of MPP Computers Used for Benchmark Calculations.

	No. of processors	Processor speed	Processor memory	Bandwidth	Latency
Daresbury SP2	26	480 Mflops / s	256 Mb	100 Mb / s	20 μs
Edinburgh T3D	512	150 Mflops / s	64 Mb	120 Mb / s	6 μs
Cineca T3E-300	128	600 Mflops / s	128 Mb	480 Mb / s	4 μs
NERSC T3E-900	512	900 Mflops / s	256 Mb	480 Mb / s	4 μs

TABLE III.
Details of Butadiene Benchmark Calculations as Carried Out in Ref. 7. All Calculations Were Done in the C_{2h} Symmetry Group and Used the Complete Active Space.

Test case	A	B
Basis set	cc-pVDZ	cc-pVTZ[a]
Size of integral file	41 Mb	495 Mb
No. orbitals	$(32a_g, 11a_u, 32b_u, 11b_g)$	$(61a_g, 26a_u, 61b_u, 26b_g)$
Reference space	4 el. in $(1-3a_u, 1-2b_g)$	
Frozen core	$(1-2a_g, 1-2b_u)$	
No. ref. config.	28	
No. contracted CSFs	199183	817599
No. uncontracted CSFs	3933377	19375033

[a] Not including the *d* functions on the hydrogen atoms.

TABLE IV.
Details of Butadiene Benchmark Calculations Carried Out on SP2. All Calculations Were Done in the C_{2h} Symmetry Group and Used the Complete Active Space.

Test case	C	D
Basis set	cc-pVDZ	aug-cc-pVDZ[a]
Size of integral file	41 Mb	121 Mb
No. orbitals	$(32a_g, 11a_u, 32b_u, 11b_g)$	$(44a_g, 17a_u, 44b_u, 17b_g)$
Reference space	4 el. in $(1-3a_u, 1-3b_g)$	6 el. in $(6-7a_g, 1-3a_u, 1-2b_g)$
Frozen core	$(1-2a_g, 1-2b_u)$	
No. ref. config.	57	142
No. contracted CSFs	344595	1569335
No. uncontracted CSFs	8590550	46994460

[a] Not including the diffuse functions on the hydrogen atoms.

TABLE V.
Details of Further Butadiene Benchmark Calculations Carried Out on T3D / T3E. All Calculations Were Carried Out in the C_{2h} Symmetry Group and Used the Complete Active Space.

Test case	E	F	G
Basis set	cc-pVTZ[a]	cc-pVTZ	
Size of integral file	495 Mb	909 Mb	
No. orbitals	$(61a_g, 26a_u, 61b_u, 26b_g)$	$(70a_g, 32a_u, 70b_u, 32b_g)$	
Reference space	4 el. in $(1-3a_u, 1-3b_g)$		6 el. in $(7-8a_g, 1-3a_u, 1-2b_g)$
Frozen core	$(1-2a_g, 1-2b_u)$		
No. ref. config.	57		142
No. contracted CSFs	1254162	1679787	3294112
No. uncontracted CSFs	42408920	59263550	144758424

[a] Not including the *d* functions on the hydrogen atoms.

TABLE VI.
Details of HOCO Benchmark Calculation Carried Out Using the C_s Symmetry Group and Used the Complete Active Space.

Test case	H
Basis set	aug-cc-pVTZ
Size of integral file	702 Mb
No. orbitals	$(106a', 55a'')$
Reference space	11 el. in $(7-12a', 1-3a'')$
Frozen core	$(1-3a')$
No. ref. config.	3048
No. contracted CSFs	10100454
No. uncontracted CSFs	1243997054

mance and are unlikely to be achieved in any scientifically productive code. (It should be noted that, while the characteristics of the T3D/T3E are quite universal, there are several different varieties of SP2s.) Not reported in this table is the memory bandwidth; this varies depending on the location of the data to be fetched, for instance, whether it is in the cache or not. The memory bandwidth of the SP2 is greater than that of T3D/T3E, so that it is easier to obtain a higher percentage of the peak performance on this machine than on the others. This then reinforces the difficulties associated with obtaining very good speedups on this machine; the large ratio of the processor speed to the interprocessor communication bandwidth, together with the rather high latency, make efficient programming of this machine a complex task.

The results shown in Tables VII and VIII compare the average times for a single iteration of the internally contracted MRCI program of MOLPRO to the uncontracted MRCI program of COLUMBUS. It is immediately obvious that the ICMRCI program is *significantly* faster. For the test case A, the single processor IBM-SP times show that the ICMRCI is, running sequentially, a factor of five times faster. For the larger test case B, taking into account the differing scaling of the two programs, we estimate that the ratio between the single pro-

TABLE VII.
Comparison of Average Time per Iteration between Contracted and Uncontracted MRCI (from Ref. 7) for Test Case A.

Number of processors	ICMRCI (MOLPRO)	MRCI (COLUMBUS)
T3D		
4	238 s	—
8	120 s	—
16	66 s	150 s
32	38 s	76 s
SP2		
1	156 s	791 s
2	114 s	401 s
4	66 s	206 s
8	39 s	104 s

TABLE VIII.
Comparison of Average Time per Iteration between Contracted and Uncontracted MRCI (from Ref. 7) for Test Case B.

Number of processors	ICMRCI (MOLPRO)	MRCI (COLUMBUS)
32	65 s	751 s
64	42 s	381 s
128	36 s	194 s

TABLE IX.
Total Calculation Times for Test Cases E and F on T3D.

Number of processors	Test case E	Test case F
32	1514 s	—
64	911 s	1109 s
128	626 s	742 s
256	—	657 s

cessor performances rises to about 10–15. It is clear that, for these test cases, the MRCI program of COLUMBUS scales very well, much better than the ICMRCI program of MOLPRO. This is of course simply due to the fact that these test cases are much too small to warrant the use of more than a couple of processors, if the use of a MPP machine at all.

Tables IX–XII present the timings of more realistic test cases, although, as an attempt is made to do the calculations with a varied number of processors, the test cases by no means approach the limit of the size of calculations possible, particularly on the T3E-900, which has a relatively large amount of memory on each processor.

The results shown in Table XI obviously do not demonstrate linear scaling, although they suggest that more than 99% of the CPU work is being done in parallel, and that the parallel efficiency on 32 processors is over 80%, around 65% on 64 processors, and falling to just below 50% on 128 processors. Table X shows that the results for the same test case (F) on the T3E-300 scale slightly less well, and Table XI for the T3E-900 demonstrates a further deterioration. This is more simply understood if we consider the considerable increase in the speed of processors from the T3D to the T3E (and seen clearly in the timings, say on 64 processors, for this test case), which although also accompanied by an increase in bandwidth and decrease in latency, are not quite matched by it. On the T3E-300

TABLE X.
Total Calculation Times for Test Cases F, G, and H on T3E-300.

Number of processors	Test case F	Test case G	Test case H
32	545 s	1376 s	6250 s
64	370 s	833 s	2985 s
128	297 s	595 s	1666 s

TABLE XI.
Total Calculation Times for Test Cases F, G, and H on T3E-900.

Number of processors	Test case F	Test case G	Test case H
16	602 s	1420 s	6113 s
32	357 s	824 s	3297 s
64	264 s	542 s	1844 s
128	—	414 s	1225 s

the results of the larger test case G, show similar scaling to test case F on the T3D, whereas those on the T3E-900 again show slightly worse scaling.

It is test case H, which has over one billion uncontracted configurations and is probably more characteristic of calculations for which the MPP ICMRCI is designed, that starts to display good speedups, particularly on the T3E-300. Unfortunately, it is not possible to run this test case on the T3D, due to the quite small amounts of memory on each processor. Although a great deal of data is distributed in the program, certain quantities, such as configuration lists and the matrices required to calculate the coupling coefficients, must be held in local memory so that the program can run efficiently.

The results for the SP2 shown in Table XII are certainly not as good as those on the T3D/T3E. The rather limited number of processors available (eight being the maximum) to run one job, makes the selection of a suitable test case difficult.

Conclusion

We have presented a fully parallel version of the MOLPRO ICMRCI program. We have demonstrated that it is possible to obtain good speedups on large problems, despite the fact there has been no major restructuring of the existing complex code, without sacrificing program portability.

TABLE XII.
Total Calculation Times for Test Cases C and D on SP2.

Number of processors	Test case C	Test case D
1	3950 s	14885 s
2	2624 s	8586 s
4	1583 s	4913 s
8	839 s	2736 s

The first stage in the parallelization involved the substitution of the fairly large amounts of disk I/O, by the reading and writing of data to global arrays, using the GA subroutine libraries. The next stage of the parallelization, in which the work was partitioned between the processors, has been carried out at the subroutine level. Several different strategies have been used, ranging from a coarse-grain decomposition of the trial vector, to a much finer-grain decomposition over the all external two-electron integrals, or intermediate configurations.

The resulting parallel program cannot be simply categorized as being either a replicated data or a distributed data code. Whereas a large proportion of the data is distributed over the processors via the use of global arrays, within each separate set of tasks, there are cases in which some of the data are distributed, but other cases in which they are not.

Acknowledgment

P. J. K. and A. J. D. are grateful to H.-J. Werner and R. J. Allan for helpful discussions. The practical assistance of Drs. R. J. Allan and G. G. Balint-Kurti is acknowledged. This research was performed in part using the Molecular Science Computing Facility in the Environmental Molecular Sciences Laboratory at the Pacific Northwest National Laboratory (PNNL).

References

1. A. J. van der Steen, *Overview of Recent Supercomputers*, National Computing Facilities Foundation, The Hague, The Netherlands, 1997.
2. J. J. Dongarra, H. Meuer, and E. Stohmaier, *Supercomputer*, **12**, 1 (1996).
3. R. J. Harrison and R. Shepard, *Ann. Rev. Phys. Chem.*, **45**, 623 (1994).
4. M. F. Guest, P. Sherwood, and J. H. van Lenthe, *Theor. Chim. Acta*, **84**, 423 (1993).
5. M. F. Guest, R. J. Harrison, J. H. van Lenthe, and L. C. H. Corler, *Theor. Chim. Acta*, **71**, 117 (1987).
6. M. Schüler, T. Konvar, H. Lischka, R. Shepard, and R. J. Harrison, *Theor. Chim. Acta*, **84**, 489 (1993).
7. H. Dachsel, H. Lischka, R. Shepard, J. Nieplocha, and R. J. Harrison, *J. Comput. Chem.*, **18**, 430 (1997).
8. R. Kobayashi and A. P. Rendell, *Chem. Phys. Lett.*, **265**, 1 (1997).
9. W. Meyer, In *Modern Theoretical Chemistry*, H. F. Schaefer III, Ed., Plenum Press, New York, 1977.
10. H.-J. Werner and E.-A. Reinsch, *J. Chem. Phys.*, **76**, 3144 (1982).
11. H.-J. Werner and P. J. Knowles, *J. Chem. Phys.*, **89**, 5803 (1988).
12. P. J. Knowles and H.-J. Werner, *Chem. Phys. Lett.*, **145**, 514 (1988).
13. H.-J. Werner, In *Ab Initio Methods in Quantum Chemistry II*, K. P. Lawley, Ed., John Wiley & Sons, New York, 1987.
14. P. J. Knowles and H.-J. Werner, *Theor. Chim. Acta*, **84**, 95 (1992).
15. MOLPRO is a package of *ab initio* programs written by H.-J. Werner and P. J. Knowles, with contributions from R. D. Amos, A. Berning, D. L. Cooper, M. J. O. Deegan, A. J. Dobbyn, F. Eckert, C. Hampel, T. Leininger, R. Lindh, A. W. Lloyd, W. Meyer, M. E. Mura, A. Nicklass, P. Palmieri, K. Peterson, R. Pitzer, P. Pulay, G. Rauhut, M. Schütz, H. Stoll, A. J. Stone, and T. Thorsteinsson.
16. J. Nieplocha, R. J. Harrison, and R. J. Littlefield, In *Proceedings of Supercomputing 1994*, IEEE Computer Society Press, Washington, DC, 1994.
17. R. J. Harrison, *Int. J. Quantum Chem.*, **40**, 847 (1991).
18. B. Roos, P. Taylor, and P. E. M. Siegbahn, *Chem. Phys.*, **48**, 157 (1980).
19. H.-J. Werner and P. J. Knowles, *J. Chem. Phys.*, **82**, 5053 (1985).
20. P. J. Knowles and H.-J. Werner, *Chem. Phys. Lett.*, **115**, 259 (1985).
21. E. R. Davidson, *J. Comput. Phys.*, **17**, 87 (1975).
22. MPI: A message passing interface standard, Message Passing Interface Forum, University of Tennessee, Knoxville, TN, June 12, 1995.